

Deferred Shading & Screen Space Effects

State of the Art Rendering Techniques used in the 3D Games Industry

Sebastian Lehmann | 11. Februar 2014

FREESTYLE PROJECT – GRAPHICS PROGRAMMING LAB – CHAIR OF COMPUTER GRAPHICS – WINTER TERM 2013 / 2014



Rendering a Complex Scene

Render time depends on:

- Number of objects N (scene complexity)
- Number of lights L
- Number of screen fragments R (resolution)

Rendering a Complex Scene

Render time depends on:

- Number of objects N (scene complexity)
- Number of lights L
- Number of screen fragments R (resolution)

Trend: increasing!

Conventional Rendering Method

for each object

- for each fragment
 - for each light
 - compute lighting

Conventional Rendering Method

for each object

- for each fragment
 - for each light
 - compute lighting

⇒ requires time $\mathcal{O}(N \cdot L \cdot R)$

Not reasonable in modern games

Deferred Shading: Two Passes

for each object

- for each fragment
 - store surface material properties in offscreen buffer

Deferred Shading: Two Passes

for each object

- for each fragment
 - store surface material properties in offscreen buffer

for each light

- for each fragment
 - fetch surface material properties from offscreen buffer
 - compute lighting
 - add to screen color

Deferred Shading: Two Passes

for each object

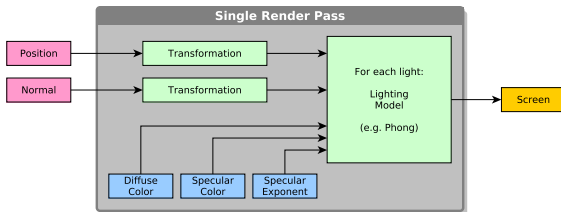
- for each fragment
 - store surface material properties in offscreen buffer

for each light

- for each fragment
 - fetch surface material properties from offscreen buffer
 - compute lighting
 - add to screen color

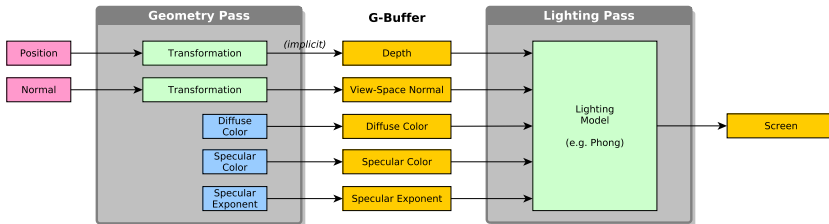
⇒ requires time $\mathcal{O}((N + L) \cdot R)$

Deferred Shading – Rendering Pipeline



(Conventional method for comparison)

Deferred Shading – Rendering Pipeline



Demo

Problem: Translucent objects

- User sees multiple objects at same pixel
- Need to evaluate lighting model multiple times
- G-Buffer can't store this information

Problem: Translucent objects

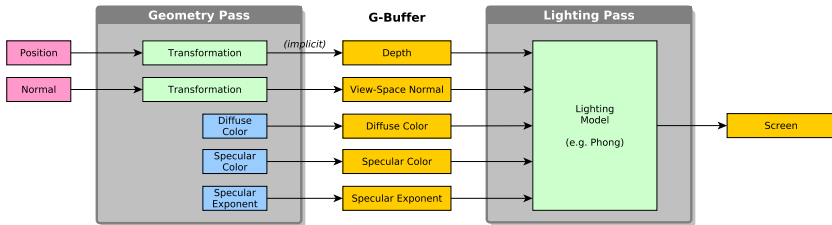
- User sees multiple objects at same pixel
- Need to evaluate lighting model multiple times
- G-Buffer can't store this information

Solution: Render them *separately*

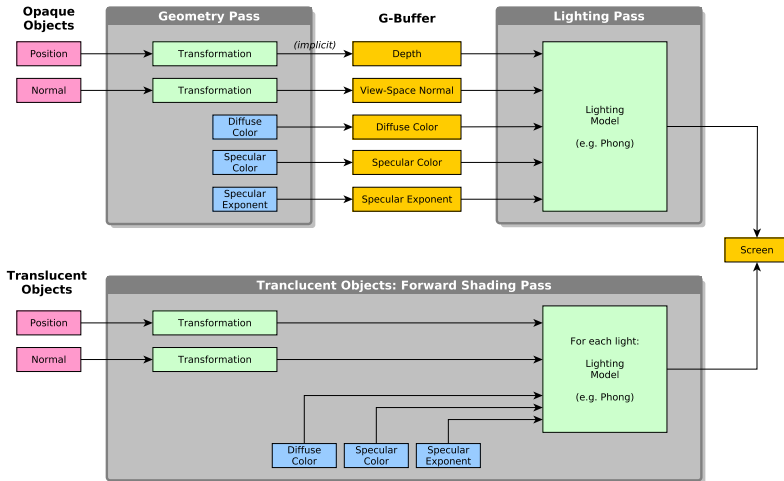
Two methods:

- *Multiple Layers*: render each layer with Deferred Shading
(Complex and costly)
- *Forward Shading*: render these objects using conventional method
(But restrict the set of lights!) My choice

Deferred Shading – Rendering Pipeline



Deferred Shading – Rendering Pipeline



Demo

Ambient Illumination

- Indirect light (“bounce”) also illuminates the scene
- Usually: Add constant illumination ($\sim 5 \dots 10 \%$) everywhere
(But unrealistic \Rightarrow need to reduce at occluded points)

Ambient Illumination

- Indirect light (“bounce”) also illuminates the scene
- Usually: Add constant illumination ($\sim 5 \dots 10 \%$) everywhere
(But unrealistic \Rightarrow need to reduce at occluded points)

Ambient Occlusion

- Do not add the same global illumination everywhere
(Have another factor influencing global illumination)
- Compute factor depending on local environment
(“How much light can reach this point?”)

Ambient Occlusion – Example

Starcraft 2



Ambient Occlusion **Disabled**



Ambient Occlusion **Enabled**

Per-Object Ambient Occlusion

- Precompute occlusion map (texture) per object
(Assumes static object mesh)
- Cheap technique
- But no *inter-object* ambient occlusion, bad for highly dynamic scenes
(But for some games almost perfect, e.g. RTS)

Per-Object Ambient Occlusion

- Precompute occlusion map (texture) per object
(Assumes static object mesh)
- Cheap technique
- But no *inter-object* ambient occlusion, bad for highly dynamic scenes
(But for some games almost perfect, e.g. RTS)

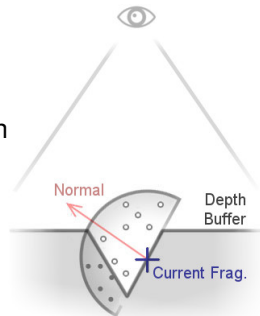
Screen-Space Ambient Occlusion (SSAO)

- Approximate “reachability” per pixel *using rendered image*
- Used in almost all modern games
(or variations of SSAO)

Ambient Occlusion – SSAO

Idea

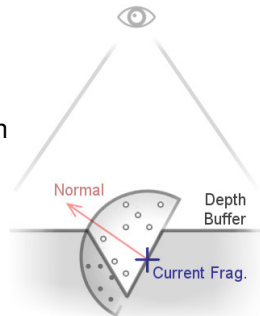
- For each pixel, cast rays to look for occlusions
- The more rays hit objects, the less light comes in
- Use depth buffer



Ambient Occlusion – SSAO

Idea

- For each pixel, cast rays to look for occlusions
- The more rays hit objects, the less light comes in
- Use depth buffer



Problems

- Many rays per pixel are inefficient
(Use only few and blur the result, but how?)
- Objects in the front: Are they occluding the scene in the back?
(Information missing)
- What about the screen border?
(Information missing)

How many rays, in which direction?

- Rays with random direction within hemisphere (2 to 8 per pixel)
- Different set of directions for each pixel
(Repeating pattern of size 4x4 to 8x8)
- Post-process: blur with *filter radius = pattern size*
(Effectively kills noise almost entirely)
- Per ray: only 2 or 3 marching steps are enough, random initial offset

How many rays, in which direction?

- Rays with random direction within hemisphere (2 to 8 per pixel)
- Different set of directions for each pixel
(Repeating pattern of size 4x4 to 8x8)
- Post-process: blur with *filter radius* = *pattern size*
(Effectively kills noise almost entirely)
- Per ray: only 2 or 3 marching steps are enough, random initial offset

Wait... `rand()` on the GPU?

- Precompute noise texture

Objects in the front

- They hide the scene behind them
- Information missing!
- *Range check*: If ray hits an object too far in the front \Rightarrow ignore ray
- Configurable threshold

This Buddah is flying in front of the background:



without range check



with range check

Screen border

- Information missing!
(Unless we rendered into enlarged framebuffer)
- *Border check*: If ray shoots outside screen \Rightarrow ignore ray
- Effectively fades out the effect at screen borders
(Almost unnoticable)

Screen border

- Information missing!
(Unless we rendered into enlarged framebuffer)
- *Border check*: If ray shoots outside screen \Rightarrow ignore ray
- Effectively fades out the effect at screen borders
(Almost unnoticable)

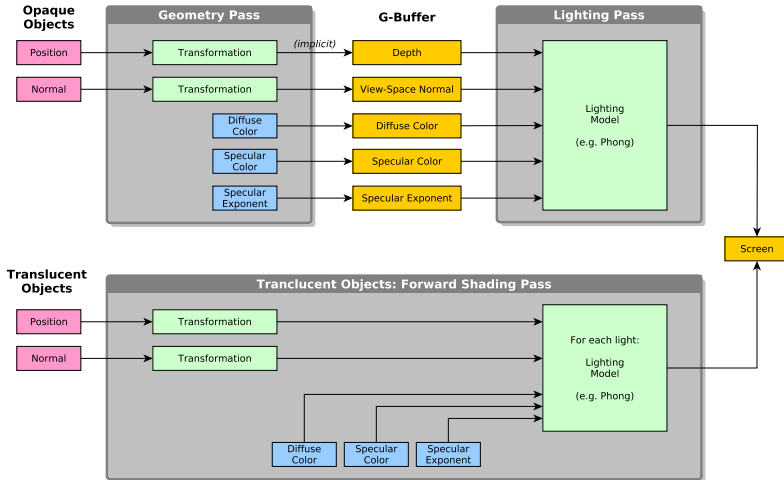
Blurring might remove details

- Don't blur over edges / stay within same surface
- Compare depth and normal at source and target
- Known as *geometry-aware* / *bilateral* filter

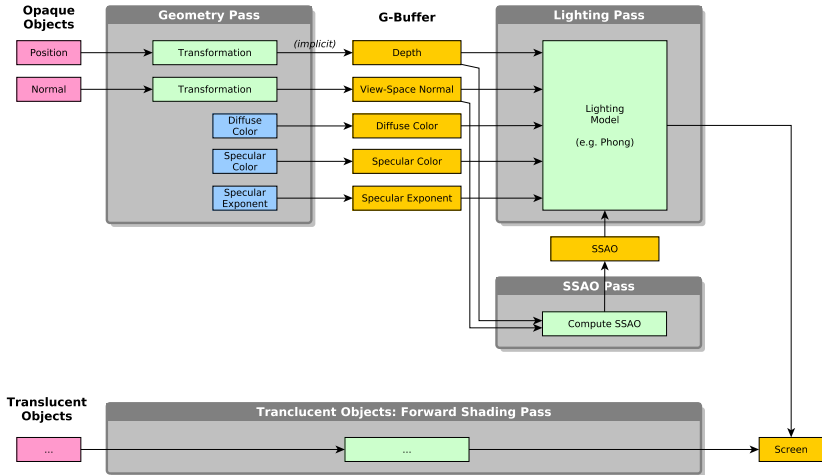


Example of bad blur!

Ambient Occlusion – SSAO – Rendering Pipeline



Ambient Occlusion – SSAO – Rendering Pipeline



Demo

How can we render (non-recursive) reflections?

- Planar mirrors: duplicate scene
(Doesn't scale. OK for single mirror like ocean.)
- Single curved reflectors: cube maps
(Doesn't scale either. Cool for cars.)
- Complex scene / general: screen space
(Scales well, but information missing. Good for short distance ("local") reflections)



Idea

- Again ray casting, now along reflection vector (~ 8 to 60 steps)
- When intersection found \Rightarrow duplicate that color
- Non-perfect reflectors: add random jitter to ray direction

Idea

- Again ray casting, now along reflection vector (~ 8 to 60 steps)
- When intersection found \Rightarrow duplicate that color
- Non-perfect reflectors: add random jitter to ray direction

Problems, problems, problems ...

- What if (real) intersection is behind occluder or outside screen?
(Information missing)
- Too large step size: we miss the intersection!
(Did we only miss it, or is something in front?)
- What if the intersection is a back face (and thus invisible)?
(Information missing)
- Should we reflect the whole color or only diffuse part?
(Remember: specular lighting is view-dependent!)

Finding the “best” intersection

- Proceed ray casting until we find pixel with lower depth
- Cancel ray casting after k steps or when outside of the screen

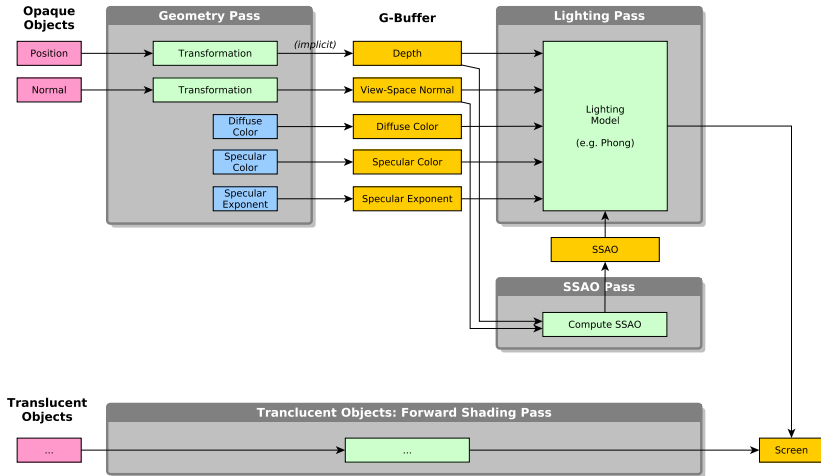
Finding the “best” intersection

- Proceed ray casting until we find pixel with lower depth
- Cancel ray casting after k steps or when outside of the screen
- For each sample which has lower depth and is front face:
 - Remember the sample with smallest depth error
 - If depth way too small (configurable tolerance):
 - Assume there is an occluder
 - Optional: “recover” from small occluders by counting these cases

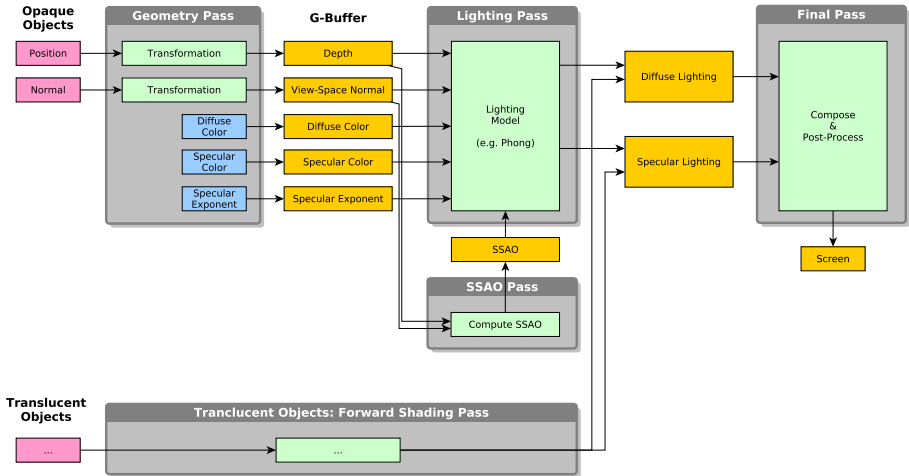
Finding the “best” intersection

- Proceed ray casting until we find pixel with lower depth
- Cancel ray casting after k steps or when outside of the screen
- For each sample which has lower depth and is front face:
 - Remember the sample with smallest depth error
 - If depth way too small (configurable tolerance):
 - Assume there is an occluder
 - Optional: “recover” from small occluders by counting these cases
- Final result is sample with smallest depth error, but
 - fade out effect when error too large
 - fade out effect when almost at screen border
 - fade out effect when almost a back face

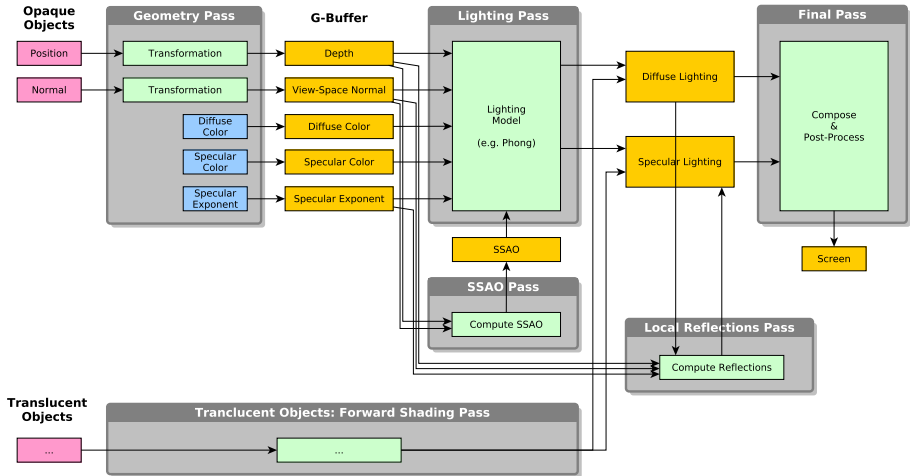
Local Reflections in SS – Rendering Pipeline



Local Reflections in SS – Rendering Pipeline



Local Reflections in SS – Rendering Pipeline



Demo